L7 - Array, conditional

February 11, 2020

0.1 Introduction to Arrays

A Java `Array` is a *fixed-size* collection. They use a special syntax (for historical reasons, so for the worst possible reasons) and have no methods. `Array` methods are provided by classes like `java.util.Arrays` and are static.

In some design cases, we will know a collection's size in advance; `Array` is useful for these cases. Arrays can store object references or primitive-type values, unlike Java's `List` types.

0.2 `weblog-analyzer` project

- records details of each access to a website
- supports analysis:
  - most popular pages
  - busiest time of day
  - delivered data volumes
  - broken references
  - performs analysis hour-by-hour; fixed number of these in a day

[5]:
```
import java.util.Calendar;

/**
 * Store the data from a single line of a
 * web-server log file.
 * Individual fields are made available via
 * accessors such as getHour() and getMinute().
 *
 * @author David J. Barnes and Michael Kölling.
 * @version    2016.02.29
 */
public class LogEntry implements Comparable<LogEntry>
{
    // Where the data values extracted from a single
    // log line are stored.
    private int[] dataValues;
    // The equivalent Calendar object for the log time.
```

```
    private Calendar when;

    // At which index in dataValues the different fields
    // from a log line are stored.
    private static final int YEAR = 0, MONTH = 1, DAY = 2,
                             HOUR = 3, MINUTE = 4;
    // The number of fields. If more fields are added, e.g. for
    // seconds or a status code, then this value must be increased
    // to match.
    private static final int NUMBER_OF_FIELDS = 5;

    /**
     * Decompose a log line so that the individual fields
     * are available.
     * @param logline A single line from the log.
     *                This should be in the format:
     *                year month day hour minute etc.
     */
    public LogEntry(String logline)
    {
        // The array to store the data for a single line.
        dataValues = new int[NUMBER_OF_FIELDS];
        // Break up the log line.
        LoglineTokenizer tokenizer = new LoglineTokenizer();
        tokenizer.tokenize(logline,dataValues);
        setWhen();
    }

    /**
     * Create a LogEntry from the individual components.
     * @param year The year
     * @param month The month (1-12)
     * @param day The day (1-31)
     * @param hour The hour (0-23)
     * @param minute The minute (0-59)
     */
    public LogEntry(int year, int month, int day, int hour, int minute)
    {
        // The array to store the data for a single line.
        dataValues = new int[NUMBER_OF_FIELDS];
        dataValues[YEAR] = year;
        dataValues[MONTH] = month;
        dataValues[DAY] = day;
        dataValues[HOUR] = hour;
        dataValues[MINUTE] = minute;
        setWhen();
    }
```

```
    /**
     * Return the hour.
     * @return The hour field from the log line.
     */
    public int getHour()
    {
        return dataValues[HOUR];
    }

    /**
     * Return the minute.
     * @return The minute field from the log line.
     */
    public int getMinute()
    {
        return dataValues[MINUTE];
    }

    /**
     * Create a string representation of the data.
     * This is not necessarily identical with the
     * text of the original log line.
     * @return A string representing the data of this entry.
     */
    public String toString()
    {
        StringBuffer buffer = new StringBuffer();
        for(int value : dataValues) {
           // Prefix a leading zero on single digit numbers.
            if(value < 10) {
                buffer.append('0');
            }
            buffer.append(value);
            buffer.append(' ');
        }
        // Drop any trailing space.
        return buffer.toString().trim();
    }

    /**
     * Compare the date/time combination of this log entry
     * with another.
     * @param otherEntry The other entry to compare against.
     * @return A negative value if this entry comes before the other.
     *         A positive value if this entry comes after the other.
     *         Zero if the entries are the same.
```

```
     */
    public int compareTo(LogEntry otherEntry)
    {
        // Use the equivalent Calendars comparison method.
        return when.compareTo(otherEntry.getWhen());
    }

    /**
     * Return the Calendar object representing this event.
     * @return The Calendar for this event.
     */
    private Calendar getWhen()
    {
        return when;
    }

    /**
     * Create an equivalent Calendar object from the data values.
     */
    private void setWhen()
    {
        when = Calendar.getInstance();
        // Adjust from 1-based month and day to 0-based.
        when.set(dataValues[YEAR],
                 dataValues[MONTH] - 1, dataValues[DAY] - 1,
                 dataValues[HOUR], dataValues[MINUTE]);
    }

}
```

[6]:
```
/**
 * Read web server data and analyse hourly access patterns.
 *
 * @author David J. Barnes and Michael Kölling.
 * @version    2016.02.29
 */
public class LogAnalyzer
{
    // Where to calculate the hourly access counts.
    private int[] hourCounts;
    // Use a LogfileReader to access the data.
    private LogfileReader reader;

    /**
     * Create an object to analyze hourly web accesses.
     */
    public LogAnalyzer()
```

```
    {
        // Create the array object to hold the hourly
        // access counts.
        hourCounts = new int[24];
        // Create the reader to obtain the data.
        reader = new LogfileReader();
    }

    /**
     * Analyze the hourly access data from the log file.
     */
    public void analyzeHourlyData()
    {
        while(reader.hasNext()) {
            LogEntry entry = reader.next();
            int hour = entry.getHour();
            hourCounts[hour]++;
        }
    }

    /**
     * Print the hourly counts.
     * These should have been set with a prior
     * call to analyzeHourlyData.
     */
    public void printHourlyCounts()
    {
        System.out.println("Hr: Count");
        for(int hour = 0; hour < hourCounts.length; hour++) {
            System.out.println(hour + ": " + hourCounts[hour]);
        }
    }

    /**
     * Print the lines of data read by the LogfileReader
     */
    public void printData()
    {
        reader.printData();
    }
}
```

[7]:
```
import java.util.Scanner;

/**
 * Break up line from a web server log file into
 * its separate fields.
```

```
 * Currently, the log file is assumed to contain simply
 * integer date and time information.
 *
 * @author David J. Barnes and Michael Kolling.
 * @version    2016.02.29
 */
public class LoglineTokenizer
{
    /**
     * Construct a LogLineAnalyzer
     */
    public LoglineTokenizer()
    {
    }

    /**
     * Tokenize a log line. Place the integer values from
     * it into an array. The number of tokens on the line
     * must be sufficient to fill the array.
     *
     * @param logline The line to be tokenized.
     * @param dataLine Where to store the values.
     */
    public void tokenize(String logline, int[] dataLine)
    {
        try {
            // Scan the logline for integers.
            Scanner tokenizer = new Scanner(logline);
            for(int i = 0; i < dataLine.length; i++) {
                dataLine[i] = tokenizer.nextInt();
            }
        }
        catch(java.util.NoSuchElementException e) {
            System.out.println("Insuffient data items on log line: " + logline);
            throw e;
        }
    }
}
```

[8]:
```
import java.io.File;
import java.io.FileNotFoundException;
import java.net.URISyntaxException;
import java.net.URL;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.Random;
```

```java
import java.util.Scanner;

/**
 * A class to read information from a file of web server accesses.
 * Currently, the log file is assumed to contain simply
 * date and time information in the format:
 *
 *    year month day hour minute
 * Log entries are sorted into ascending order of date.
 *
 * @author David J. Barnes and Michael Kölling.
 * @version    2016.02.29
 */
public class LogfileReader implements Iterator<LogEntry>
{
    // The data format in the log file.
    private String format;
    // Where the file's contents are stored in the form
    // of LogEntry objects.
    private ArrayList<LogEntry> entries;
    // An iterator over entries.
    private Iterator<LogEntry> dataIterator;

    /**
     * Create a LogfileReader to supply data from a default file.
     */
    public LogfileReader()
    {
        this("weblog.txt");
    }

    /**
     * Create a LogfileReader that will supply data
     * from a particular log file.
     * @param filename The file of log data.
     */
    public LogfileReader(String filename)
    {
        // The format for the data.
        format = "Year Month(1-12) Day Hour Minute";
        // Where to store the data.
        entries = new ArrayList<>();

        // Attempt to read the complete set of data from file.
        boolean dataRead;
        try{
            // Locate the file with respect to the current environment.
```

```
            URL fileURL = getClass().getClassLoader().getResource(filename);
            if(fileURL == null) {
                throw new FileNotFoundException(filename);
            }
            Scanner logfile = new Scanner(new File(fileURL.toURI()));
            // Read the data lines until the end of file.
            while(logfile.hasNextLine()) {
                String logline = logfile.nextLine();
                // Break up the line and add it to the list of entries.
                LogEntry entry = new LogEntry(logline);
                entries.add(entry);
            }
            logfile.close();
            dataRead = true;
        }
        catch(FileNotFoundException | URISyntaxException e) {
            System.out.println("Problem encountered: " + e);
            dataRead = false;
        }
        // If we couldn't read the log file, use simulated data.
        if(!dataRead) {
            System.out.println("Failed to read the data file: " + filename);
            System.out.println("Using simulated data instead.");
            createSimulatedData(entries);
        }
        // Sort the entries into ascending order.
        Collections.sort(entries);
        reset();
    }

    /**
     * Does the reader have more data to supply?
     * @return true if there is more data available,
     *         false otherwise.
     */
    public boolean hasNext()
    {
        return dataIterator.hasNext();
    }

    /**
     * Analyze the next line from the log file and
     * make it available via a LogEntry object.
     *
     * @return A LogEntry containing the data from the
     *         next log line.
     */
```

```
    public LogEntry next()
    {
        return dataIterator.next();
    }

    /**
     * Remove an entry.
     * This operation is not permitted.
     */
    public void remove()
    {
        System.err.println("It is not permitted to remove entries.");
    }

    /**
     * @return A string explaining the format of the data
     *         in the log file.
     */
    public String getFormat()
    {
        return format;
    }

    /**
     * Set up a fresh iterator to provide access to the data.
     * This allows a single file of data to be processed
     * more than once.
     */
    public void reset()
    {
        dataIterator = entries.iterator();
    }

    /**
     * Print the data.
     */
    public void printData()
    {
        for(LogEntry entry : entries) {
            System.out.println(entry);
        }
    }

    /**
     * Provide a sample of simulated data.
     * NB: To simplify the creation of this data, no
     * days after the 28th of a month are ever generated.
```

```
     * @param data Where to store the simulated LogEntry objects.
     */
    private void createSimulatedData(ArrayList<LogEntry> data)
    {
        LogfileCreator creator = new LogfileCreator();
        // How many simulated entries we want.
        int numEntries = 100;
        for(int i = 0; i < numEntries; i++) {
            data.add(creator.createEntry());
        }
    }
}
```

[9]:
```
import java.io.*;
import java.util.*;

/**
 * A class for creating log files of random data.
 *
 * @author David J. Barnes and Michael Kölling
 * @version    2016.02.29
 */
public class LogfileCreator
{
    private Random rand;

    /**
     * Create log files.
     */
    public LogfileCreator()
    {
        rand = new Random();
    }

    /**
     * Create a file of random log entries.
     * @param filename The file to write.
     * @param numEntries How many entries.
     * @return true if successful, false otherwise.
     */
    public boolean createFile(String filename, int numEntries)
    {
        boolean success = false;

        if(numEntries > 0) {
            try (FileWriter writer = new FileWriter(filename)) {
                LogEntry[] entries = new LogEntry[numEntries];
```

```
                for(int i = 0; i < numEntries; i++) {
                    entries[i] = createEntry();
                }
                Arrays.sort(entries);
                for(int i = 0; i < numEntries; i++) {
                    writer.write(entries[i].toString());
                    writer.write('\n');
                }

                success = true;
            }
            catch(IOException e) {
                System.err.println("There was a problem writing to " +␣
↪filename);
            }

        }
        return success;
    }

    /**
     * Create a single (random) entry for a log file.
     * @return A log entry containing random data.
     */
    public LogEntry createEntry()
    {
        int year = 2016;
        int month = 1 + rand.nextInt(12);
        // Avoid the complexities of days-per-month.
        int day = 1 + rand.nextInt(28);
        int hour = rand.nextInt(24);
        int minute = rand.nextInt(60);
        return new LogEntry(year, month, day, hour, minute);
    }

}
```

0.2.1 Creating an array object

See `public LogAnalyser(): hourCounts = new int[24]` creates an array of 24 integers. The array size must be specified at the time of creation.

0.2.2 Using the array object

Square bracket notation accesses objects in the array:

- declaration: `private String[] names`
  - ARE YOU KIDDING ME WITH THIS

  - CAN YOU NOT JUST PICK ONE SYNTAX YOU MASSIVE KNOBS
- declaration with initialization: 'private int[] spaghetti = {1, 1, 1};
  - immutable size is inferred from the declared array
- access: `y = hourCount[3]`
- assign: `hourCount[2] = (int) 4`
- in expressions: `if ( hourCount[hour] > 9 )...`
- getting array length: `int touchda = spaghetti.length;`
  - no parentheses; length is a *field*, not a method
    * AGAIN COULD YOU JUST NOT

0.3 `for` Loop

General syntax:

```
for ( initialization; condition; post-body action )
{
    statement to do
}
```

Equivalent to:

```
while ( condition )
{
    statement to do
    post-body action
}
```

Iterating through an array

```
for (int hour = 0; hour < hourCounts.length; hour++)
{
    System.out.println(hour + ": " + hourCounts[hour]);
}
```

0.4 Array-Related Methods

- `System` has static `ArrayCopy`
- `java.util.Arrays` has methods for sorting, etc
  - `binarySearch`
  - `fill` - fill the entire array with a value
  - `sort`
- `ArrayList` has `toArray` for type conversion

[12]:
```
// practice

public int[] numbers = { 4, 1, 22, 9, 14, 3, 9};

for (int i = 0; i < numbers.length; i++)
{
    System.out.println( numbers[i] );
}
```

```
4
1
22
9
14
3
9
```

[17]:
```
// fibonacci
int howMany = 20;

int [] fib = new int[howMany];

fib[0] = 0;
fib[1] = 1;

// calculate fibonacci sequence
for ( int i = 2; i < fib.length; i++ )
{
    fib[i] = fib[i-1] + fib[i-2];
}

// printout the numbers
for (int i = 0; i < fib.length; i++)
{
    System.out.println( fib[i] );
}
```

```
0
1
1
2
3
5
8
13
21
34
55
```

```
89
144
233
377
610
987
1597
2584
4181
```

0.5 `for` loop with an iterator

Note that we can leave the post-body action blank since the `iterator()` object will provide use with the means to advance to the next object.

[ ]:
```
for ( Iterator<Track> it = tracks.iterator(); it.hasNext(); )
{
    Track track = it.next() // moves to the next object
}
```

0.6 Cellular Automaton Project

An array of cells; which each maintain a simple state ("alive"/"dead"). These states change according to simple rules, based on neighbouring states.

- in automaton-v1:
  - new state: `nextState[i] = (state[i-1] + state[i] + state[i+1]) %2`

0.6.1 Conditional operator

Choose between two values: `condition ? value1 : value2`

- if `condition` is true, returns `value1`
- if `condition` is false, returns `value2`

Hot take: if you want to write concise code, don't write it in Java FFS

[19]:
```
// trivial case
public int state[] = {0, 1, 0, 1, 1, 0, 1, 0};

for (int cellValue : state)
{
    System.out.print(cellValue == 1 ? '+' : ' ');
}
```

```
 + ++ +
```

0.6.2 The Automaton Class

[20]:
```
import java.util.Arrays;

/**
 * Model a 1D elementary cellular automaton.
 *
 * @author David J. Barnes and Michael Kölling
 * @version  2016.02.29 - version 1
 */
public class Automaton
{
    // The number of cells.
    private final int numberOfCells;
    // The state of the cells.
    private int[] state;

    /**
     * Create a 1D automaton consisting of the given number of cells.
     * @param numberOfCells The number of cells in the automaton.
     */
    public Automaton(int numberOfCells)
    {
        this.numberOfCells = numberOfCells;
        state = new int[numberOfCells];
        // Seed the automaton with a single 'on' cell in the middle.
        state[numberOfCells / 2] = 1;
    }

    /**
     * Print the current state of the automaton.
     */
    public void print()
    {
        for(int cellValue : state) {
            if(cellValue == 1) {
                System.out.print("*");
            }
            else {
                System.out.print(" ");
            }
        }
        System.out.println();
    }

    /**
     * Update the automaton to its next state.
     */
```

```
    public void update()
    {
        // Build the new state in a separate array.
        int[] nextState = new int[state.length];
        // Naively update the state of each cell
        // based on the state of its two neighbors.
        for(int i = 0; i < state.length; i++) {
            int left, center, right;
            if(i == 0) {
                left = 0;
            }
            else {
                left = state[i - 1];
            }
            center = state[i];
            if(i + 1 < state.length) {
                right = state[i + 1];
            }
            else {
                right = 0;
            }
            nextState[i] = (left + center + right) % 2;
        }
        state = nextState;
    }

    /**
     * Reset the automaton.
     */
    public void reset()
    {
        Arrays.fill(state, 0);
        // Seed the automaton with a single 'on' cell.
        state[numberOfCells / 2] = 1;

    }
}
```

Running the automaton

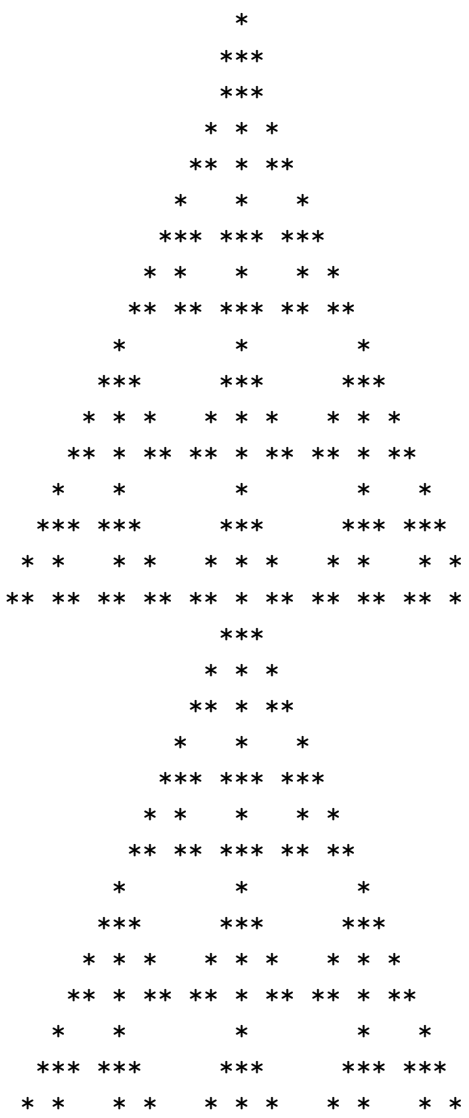
[25]:
```
Automaton auto = new Automaton(30);
```

[29]:
```
auto.reset();
auto.print();

for ( int k = 0; k < 30; k++ )
{
```

```
    auto.update();
    auto.print();

    if ( k % 15 == 0 )
    {
        auto.reset();
    }
}
```



Cool.

0.6.3 Midterm BS

Note that `ArrayLists` and control structures are *on the midterm*.

- Crib sheet may be double-sided; but must be handwritten

- fix the broken code
- implement the unwritten function
- nothing to invent
- allowed water, tea, coffee

[ ]: